由于 C++ 语言没有自动内存回收机制，程序员每次 new 出来的内存都要手动 delete。程序员忘记 delete，流程太复杂，最终导致没有 delete，异常导致程序过早退出，没有执行 delete 的情况并不罕见。

用智能指针便可以有效缓解这类问题，本文主要讲解std::auto\_ptr智能指针。

对于编译器来说，智能指针实际上是一个栈对象，并非指针类型，在栈对象生命期即将结束时，智能指针通过析构函数释放有它管理的堆内存。所有智能指针都重载了“operator->”操作符，直接返回对象的引用，用以操作对象。访问智能指针原来的方法则使用“.”操作符。

访问智能指针包含的裸指针则可以用 get() 函数。由于智能指针是一个对象，所以if (my\_smart\_object)永远为真，要判断智能指针的裸指针是否为空，需要这样判断：if (my\_smart\_object.get())。

智能指针包含了 reset() 方法，如果不传递参数（或者传递 NULL），则智能指针会释放当前管理的内存。如果传递一个对象，则智能指针会释放当前对象，来管理新传入的对象。

测试1

#include <iostream>

#include <memory>

class Simple {

public:

Simple(int param = 0) {

number = param;

std::cout << "Simple: " << number << std::endl;

}

~Simple() {

std::cout << "~Simple: " << number << std::endl;

}

void PrintSomething() {

std::cout << "PrintSomething: " << info\_extend.c\_str() << std::endl;

}

std::string info\_extend;

int number;

};

void TestAutoPtr() {

std::auto\_ptr<Simple> my\_memory(new Simple(1)); // 创建对象，输出：Simple：1

if (my\_memory.get()) { // 判断智能指针是否为空

my\_memory->PrintSomething(); // 使用 operator-> 调用智能指针对象中的函数

my\_memory.get()->info\_extend = "Addition"; // 使用 get() 返回裸指针，然后给内部对象赋值

my\_memory->PrintSomething(); // 再次打印，表明上述赋值成功

(\*my\_memory).info\_extend += " other"; // 使用 operator\* 返回智能指针内部对象，然后用“.”调用智能指针对象中的函数

my\_memory->PrintSomething(); // 再次打印，表明上述赋值成功

}

}

int main ()

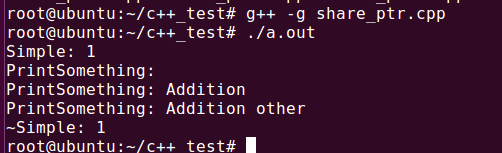
{

TestAutoPtr();

return 0;

}

编译执行，程序执行正常



测试2

std::auto\_ptr 属于 STL，当然在 namespace std 中，包含头文件 #include 便可以使用。std::auto\_ptr 能够方便的管理单个堆内存对象。

#include <iostream>

#include <memory>

class Simple {

public:

Simple(int param = 0) {

number = param;

std::cout << "Simple: " << number << std::endl;

}

~Simple() {

std::cout << "~Simple: " << number << std::endl;

}

void PrintSomething() {

std::cout << "PrintSomething: " << info\_extend.c\_str() << std::endl;

}

std::string info\_extend;

int number;

};

/\*

void TestAutoPtr() {

std::auto\_ptr<Simple> my\_memory(new Simple(1)); // 创建对象，输出：Simple：1

if (my\_memory.get()) { // 判断智能指针是否为空

my\_memory->PrintSomething(); // 使用 operator-> 调用智能指针对象中的函数

my\_memory.get()->info\_extend = "Addition"; // 使用 get() 返回裸指针，然后给内部对象赋值

my\_memory->PrintSomething(); // 再次打印，表明上述赋值成功

(\*my\_memory).info\_extend += " other"; // 使用 operator\* 返回智能指针内部对象，然后用“.”调用智能指针对象中的函数

my\_memory->PrintSomething(); // 再次打印，表明上述赋值成功

}

}

\*/

void TestAutoPtr2() {

std::auto\_ptr<Simple> my\_memory(new Simple(1));

if (my\_memory.get()) {

std::auto\_ptr<Simple> my\_memory2; // 创建一个新的 my\_memory2 对象

my\_memory2 = my\_memory; // 复制旧的 my\_memory 给 my\_memory2

my\_memory2->PrintSomething(); // 输出信息，复制成功

my\_memory->PrintSomething(); // 崩溃

}

}

int main ()

{

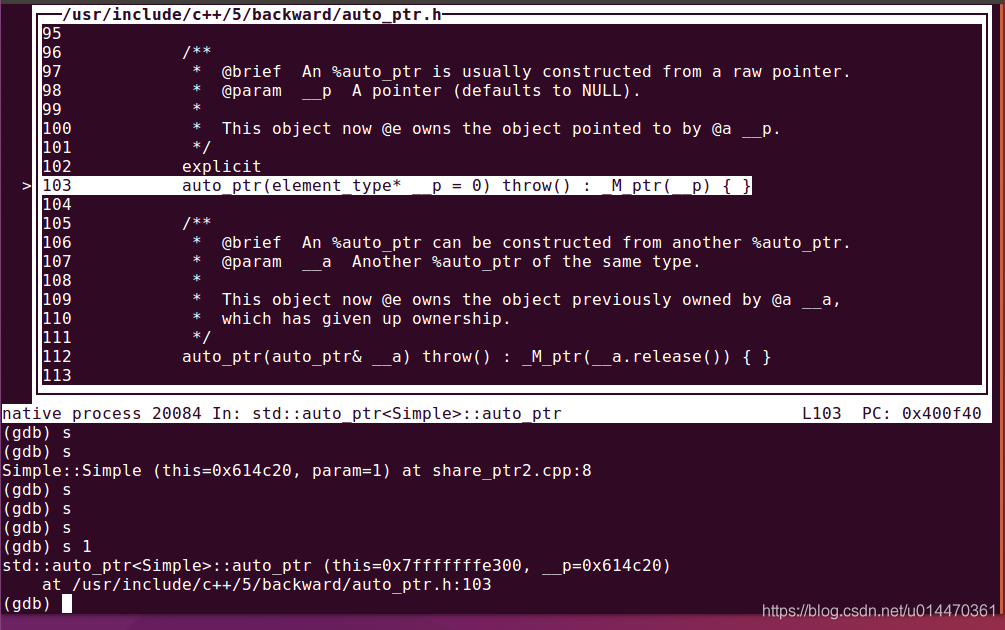
TestAutoPtr2();

return 0;

}

编译，用GDB来调试查看。

std::auto\_ptr my\_memory(new Simple(1));执行时：



std::auto\_ptr执行时解析：

/\*\*

\* 构造函数，将auto\_ptr绑定到指针\_\_p。

\* \_\_p是一个指向new出来的对象的指针，默认为0（NULL）是说auto\_ptr的构造函数可以不传参构造，

\* 这时成员\_M\_ptr=0，如果接着解引用auto\_ptr对象，将Segmentation fault。当然，通常应用auto\_ptr的构造

\* 函数会传参的。auto\_ptr提供了get函数来判断\_M\_ptr是否为空、reset函数重置\_M\_ptr指针。

\* 在继承情况下，\_M\_ptr可以是\_\_p的基类型。

\* 构造函数声明为explicit表示禁止参数的自动类型转换（因为它们总是邪恶的）。

\*

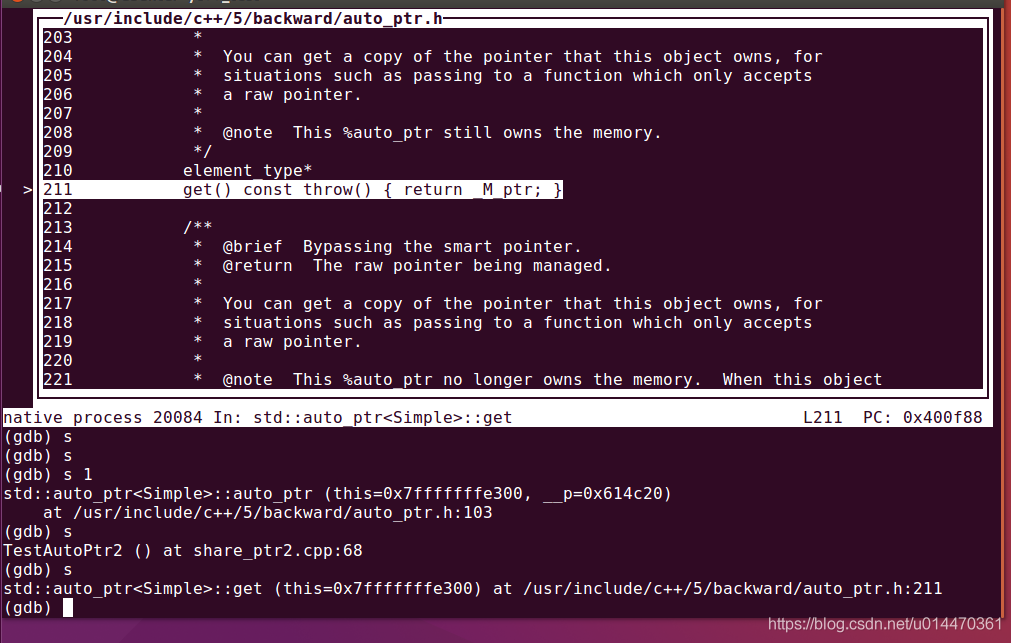
\*/

explicit auto\_ptr(element\_type\* \_\_p = 0) throw () :

\_M\_ptr(\_\_p) {

}

my\_memory.get()；执行时：



get()函数调用解析：

/\*\*

\* 返回auto\_ptr管理的指针，这通常用于判断指针是否为空的情况，所以，如果要判断

\* auto\_ptr管理的指针是否为空，不要使用if(auto\_ptr\_obj){}而是使用get函数(实际上，

\* 因为auto\_ptr并没用定义指向element\_type的dumb指针的隐式类型转换操作符，所以根本

\* 编译不过if(auto\_ptr\_obj))。

\* 但是，auto\_ptr并没有禁止你进一步操作你得到的指针，甚至delete它使

\* auto\_ptr对象内置的指针悬空。

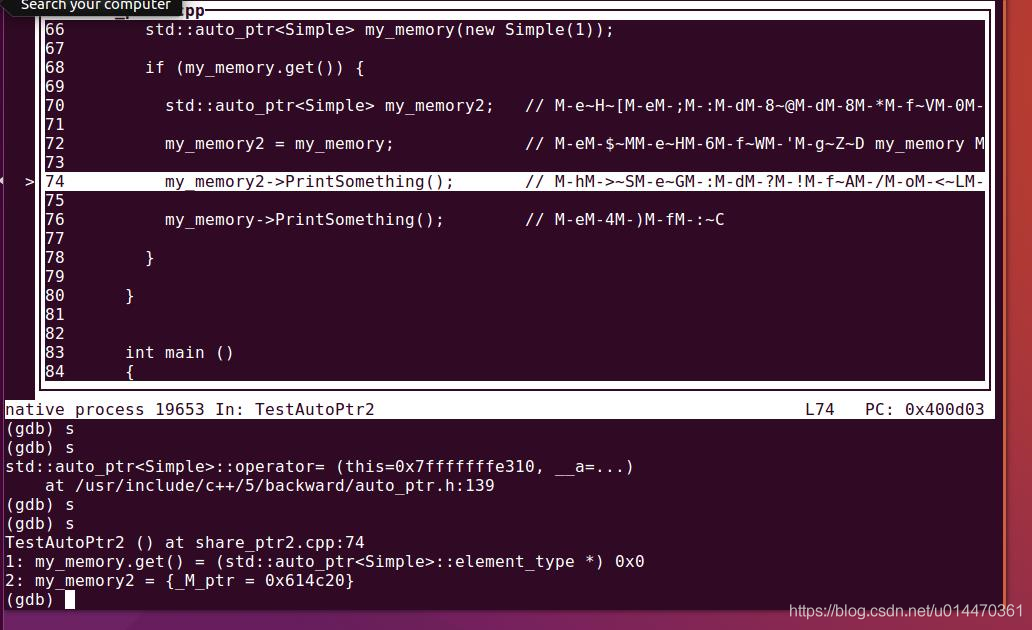
\*/

element\_type\* get() const throw () {

return \_M\_ptr;

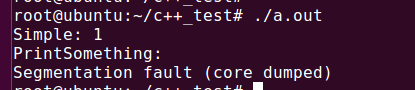
}

my\_memory2 = my\_memory; 执行时：



可以看到执行 my\_memory2 = my\_memory这句话后，my\_memory的值传给my\_memory2后，my\_memory被清零，让出了内存管理权，my\_memory2 完全夺取了 my\_memory 的内存管理所有权。

后续调用my\_memory->PrintSomething(); 时进行空指针的引用，程序报错了：Segmentation fault (core dumped)



最终如上代码导致崩溃，如上代码时绝对符合 C++ 编程思想的，居然崩溃了，跟进std::auto\_ptr 的源码后，我们看到，罪魁祸首是“my\_memory2 = my\_memory”，这行代码，my\_memory2 完全夺取了 my\_memory 的内存管理所有权，导致 my\_memory 悬空，最后使用时导致崩溃。

所以，使用 std::auto\_ptr 时，绝对不能使用“operator=”操作符。作为一个库，不允许用户使用，确没有明确拒绝[1]，多少会觉得有点出乎预料。

测试3

#include <iostream>

#include <memory>

class Simple {

public:

Simple(int param = 0) {

number = param;

std::cout << "Simple: " << number << std::endl;

}

~Simple() {

std::cout << "~Simple: " << number << std::endl;

}

void PrintSomething() {

std::cout << "PrintSomething: " << info\_extend.c\_str() << std::endl;

}

std::string info\_extend;

int number;

};

void TestAutoPtr3() {

std::auto\_ptr<Simple> my\_memory(new Simple(1));

if (my\_memory.get()) {

my\_memory.release();

}

}

int main ()

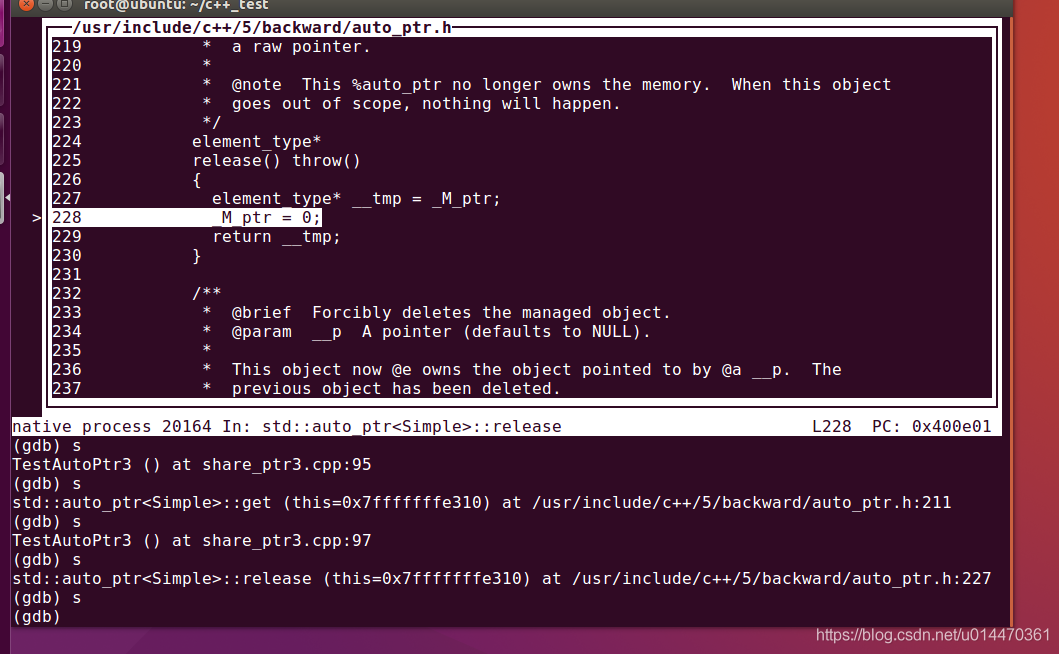
{

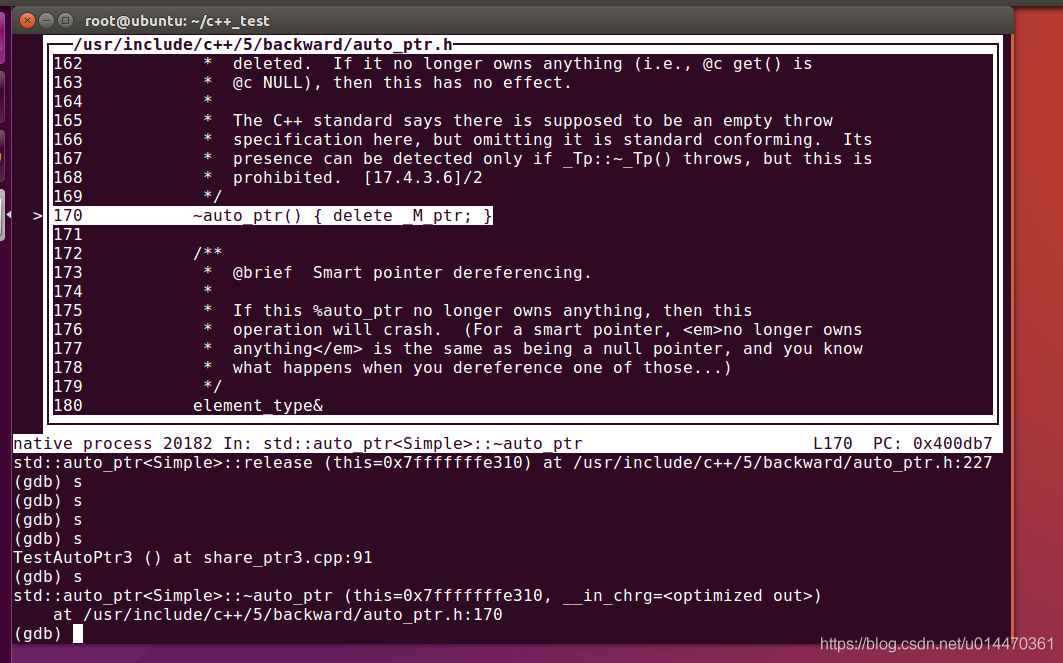
TestAutoPtr3();

return 0;

}

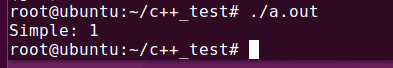
编译调试





可以看到my\_memory.release();这一句中执行时，先将指针置位0，然后delete 这个执行，实际创建的对象被没有释放掉，即对象没有被析构，没有输出“~Simple: 1”。这就造成了内存的浪费。

虽然这次执行正常退出了，但多次执行，可能会把内存搞崩溃了。



如果程序没有通过delete、free语句现实释放内存的话，不论是崩溃（崩溃前为执行释放语句）还是正常退出（即没有在程序中写下释放语句，但程序‘正常’退出）。那么这没有回收的内存将在你每一次关机时又系统回收。

这是一个致命的bug：加入程序持续运行多次，则可能在后面的某一此中因内存溢出而崩溃，而可能根本找不到原因（因为前几次都是正确的）。

当我们不想让 my\_memory 继续生存下去，我们调用 release() 函数释放内存，结果却导致内存泄露（在内存受限系统中，如果my\_memory占用太多内存，我们会考虑在使用完成后，立刻归还，而不是等到 my\_memory 结束生命期后才归还）。

测试4

#include <iostream>

#include <memory>

class Simple {

public:

Simple(int param = 0) {

number = param;

std::cout << "Simple: " << number << std::endl;

}

~Simple() {

std::cout << "~Simple: " << number << std::endl;

}

void PrintSomething() {

std::cout << "PrintSomething: " << info\_extend.c\_str() << std::endl;

}

std::string info\_extend;

int number;

};

void TestAutoPtr3() {

std::auto\_ptr<Simple> my\_memory(new Simple(1));

if (my\_memory.get()) {

Simple\* temp\_memory = my\_memory.release();

delete temp\_memory;

}

}

void TestAutoPtr4() {

std::auto\_ptr<Simple> my\_memory(new Simple(1));

if (my\_memory.get()) {

my\_memory.reset(); // 释放 my\_memory 内部管理的内存

}

}

int main ()

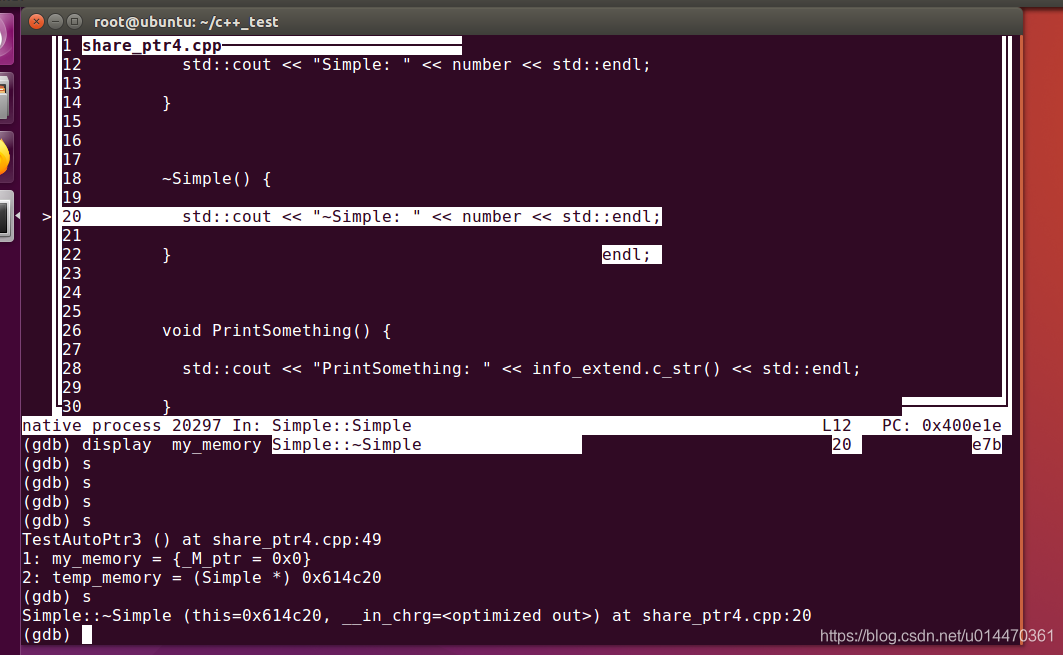
{

TestAutoPtr3();

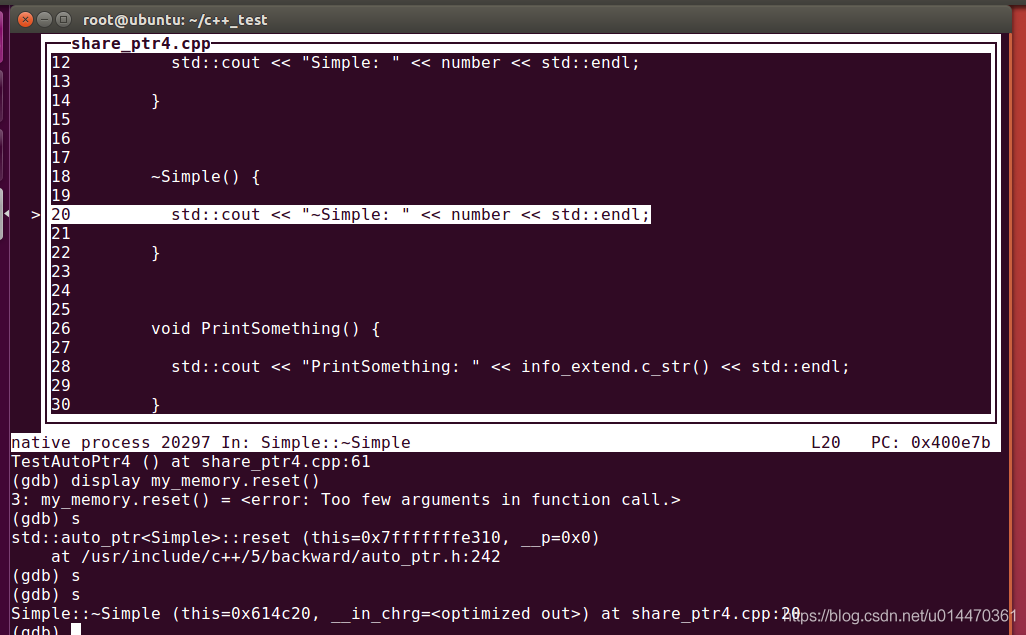
TestAutoPtr4();

return 0;

}



delete temp\_memory;执行时，调用了析构函数来释放对象的内存空间。



my\_memory.reset(); 执行时，调用了析构函数来释放对象的内存空间。

总结

原来 std::auto\_ptr 的 release() 函数只是让出内存所有权，这显然也不符合 C++ 编程思想。

总结：std::auto\_ptr 可用来管理单个对象的对内存，但是，请注意如下几点：

（1） 尽量不要使用“operator=”。如果使用了，请不要再使用先前对象。

（2） 记住 release() 函数不会释放对象，仅仅归还所有权。

（3） std::auto\_ptr 最好不要当成参数传递（读者可以自行写代码确定为什么不能）。

（4） 由于 std::auto\_ptr 的“operator=”问题，有其管理的对象不能放入 std::vector等容器中。